Problems of testing 64-bit applications

Andrey Karpov
OOO "Program Verification Systems"
Evgeniy Ryzhkov
OOO "Program Verification Systems"

March 2007

Abstract

The article observes some questions related to testing the 64-bit software. Some difficulties which a developer of resource-intensive 64-bit applications may face and the ways to overcome them are described.

The size of an average computer program increases every year. Programs become more and more complicated and knotty, process larger data sizes, get more functional and nice graphical interface. If some time earlier a program of some Kb possessing the simplest editing abilities was considered to be a full text editor, now some text editors steal tens and hundreds of Mb providing much better functionality. Naturally the requirements to the productivity of the device part of computer techniques grow with the same speed.

The next step to increase calculating power becomes the use of 64-bit microprocessor systems. This step cannot be called a revolutionary one but it allows to extend the possibilities of computer systems greatly. First of all, 64-bit systems allowed to overcome the barrier of 4Gb which had begun to limit many software developers. First of all, this concerns the developers of computational modeling packets, 3D-editors, databases, games. Large size of RAM extends the abilities of applications greatly, allowing to store large data sizes and to address them directly without loading from external data warehouses. One should remember also about higher productivity of 64-bit versions of programs which is caused by a greater number of registers, extended abilities of floating point arithmetic, the possibility of working with 64-bit numbers.

Naturally, complication of program solutions makes more complicated the task of their maintenance and testing. The impossibility of manual testing of large program systems caused the development of systems of automatization of testing and quality control of programs. There are different approaches to the providing of the necessary quality of programs and we'll remember them briefly.

The oldest, the safest and the most approved approach to defects searching is code review. [1] This method is based on the combined reading of the code with following some rules and recommendations. This way is described very well, for example, in Steve McConnell's book "Code Complete" [2]. Unfortunately, this practice is unusable for large testing of contemporary program systems due to their large size. Although this means gives the best results it is not always used in conditions of contemporary life-cycles of software development, where the terms of development and release of the product on the market is an important aspect. That's why the code review often turns into rare meetings the aim of which is to teach new and less experienced workers to write quality code rather than to check the efficiency of some modules. This is a good means to raise skills level of programmers but it cannot be considered to be a full means of controlling the quality of a program under development.

Means of static analysis of the code serve to help the developers who realize the necessity of regular code review but don't have enough time for this [3]. Their main aim is to reduce the code size which demands the human attention and by this reduce the time of the review. To static code analyzers many programs are referred which are created for different programming languages and have a large set of functions from the simplest control of code alignment to complicated analysis of potentially dangerous places. Systematic use of static analyzers allows to improve the quality of the code greatly and to find many

errors. The static-analysis approach has a lot of followers and many interesting works are devoted to it (for example [4, 5]). The advantage of this approach is that it can be used not depending on the complexity and size of a program solution under development.

There is one more means of software products quality increase which is worth attention, it's the select testing method. The base of this method is a well-known and intuitively clear means of testing of only those parts of the software which were directly affected with alterations. The main problem of application of the select testing method is the obtaining of a reliable list of all the software parts affected with alterations. Select testing method which is for example supported by a software product Testing Relief (http://www.testingrelief.net) solves this problem.

The method of white-box test [6]. Under the method of white-box test we'll understand the fulfillment of the maximum accessible number of code branches using the debugger or other means. The more code coverage the fuller the testing is. Sometimes under the testing according to the white-box test method simple debugging of the application with the search of a known error is understood. The full testing with the help of white-box test method of the whole program code has become impossible long ago due to the large size of contemporary programs. Now the white-box test method is convenient to be used on that step when the error is found and you need to understand what has caused it. The white-box test method has its opponents who deny the profit of debugging programs in real time. The main argument is that the possibility to follow the program work and to make changes in it simultaneously turns out to be a wrong approach in programming being based on a large number of code corrections by cut and try method. We won't touch upon these debates but will mention that the white-box test is a very expensive way to improve the quality of large and complicated program systems.

The black-box test method was approved much greatly [7]. Unit test may be referred to this sphere too [8]. The main idea consists in writing a set of tests for separate modules and functions, which checks all the main modes of their work. Some sources refer unit test to the white-box test method as far as it is based on knowing the program structure. The author believes that the functions and modules being tested should be considered as black-box for unit tests should not take into account the inner structure of the function. Justification for this is the methodology of development when the tests are developed before writing functions themselves, and that contributes to the increase of the control of their functionality from the point of view of specification.

A large amount of literature is devoted to unit test method, for example [9, 10]. Unit test proved while developing simple projects and difficult ones as well. One of the advantages of unit test is that it may easily check the correctness of the corrections being made in the program right in the process of developing. One tries to make so that all the tests last some minutes and that allows the developer who made corrections in the code, notice an error and correct it at once. If the run of all the tests is impossible lengthy tests are brought separately and launched at night, for example. This also serves the quick detection of errors on the next morning at least.

Manual testing. This is perhaps the final step of any development, but it should not be considered to be a good and reliable method. Manual testing must exist necessarily for it is impossible to find all the errors in automatic mode or during code review. But one should not rely on this method completely. If the program has low quality and a lot of inner defects its testing and correction may be prolonged for a long time and still you cannot provide the proper quality of the program. The only method to get a quality program is the quality code. That's why we won't observe manual testing as a full method used during the development of large projects either.

So, what is left for us to pay attention to while developing large program systems? This is static analysis and unit test. These approaches can improve the quality and safety of the program code greatly, and we should pay the largest attention to them, but of course do not forget about others.

Now let's turn to the question of testing 64-bit programs for the use of the methods we have chosen faces

some unpleasant difficulties. Let's begin with static code analyzers.

Being strange enough, in spite all its great abilities, large period of development and use practice static analyzers turned out to be poorly ready for error search in 64-bit programs. Let's examine the situation by the example of C++ code analysis as a sphere where static analyzers are used most frequently. Many static analyzers support some rules related to the search of the code which shows incorrect behavior while being ported on 64-bit systems. But they fulfill this in rather separated methods and very incompletely. It became clear after the beginning of mass development of applications for the 64-bit version of OS Windows in the environment Microsoft Visual C++ 2005.

The explanation of this is that most checks are based on rather old reference materials devoted to the investigation of the problems of program port on 64-bit systems from the point of view of C++ language. As a result some constructions which have appeared in C++ language were not paid the proper attention concerning the point of view of portability control and didn't find their use in analyzers. Some other changes were not taken into account, as for example, RAM size which has grown greatly, and the use of different data models in different compilers (LP64, LLP64, ILP64 [11]).

To make it clear let's examine two examples.

```
double *DoubleArray;
unsigned Index = 0;
while (...)
  DoubleArray[Index++] = 1.0f;
```

You won't get a warning message on such code even using so powerful analyzers as Parasoft C++test (http://www.parasoft.com) and Gimpel Software PC-Lint (http://www.gimpel.com). It is not surprising. This code does not arouse suspicion of a common developer who is used to the practice of using int or unsigned types of variables as indices. Unfortunately, this code will be inefficient on the 64-bit system if the size of processed array DoubleArray excesses size of 4GB items. In this case variable Index overflow will occur and the result of program work will be incorrect. The correct variant is to use size_t type while programming under Windows x64 (data model LLP64) or size_t/unsigned long type while programming under Linux (data model LP64).

The cause why static analyzers cannot diagnose such code is perhaps that when the questions of porting on 64-bit systems were discussed hardly anyone could imagine array with more than 4 billions items. And 4 billions of items of double type is 4 * 8 = 32 GB of memory for one array. So it is the great size, especially if we remember that it was 1993-1995. Right in that time many publications and discussions, devoted to the use of 64-bit systems, were printed.

The result may be that nobody paid attention to the possible incorrect indexing while using int type, and further the questions of port arouse rather rarely. Actually, no one static analyzer will show a warning message on the given code. An exception is perhaps only Viva64 analyzer (http://www.viva64.com). It was developed to compensate the gaps in diagnosis of 64-bit C/C++ code by other analyzers, and is based on the researches done anew. But it has a significant disadvantage which consists in that it is not the general-purpose analyzer. It concentrates only on the analysis of errors occurring while porting the code on 64-bit Windows systems and that's why can be used only combined with other analyzers to provide the proper quality of the code.

Let's examine another example.

```
char *p;
long g=(long)p;
```

With the help of this simple error you can check which data models the static analyzer you use can

understand. The problem is that most of them are intended for data model LP64. It is caused by the history of development of 64-bit systems too. It is data model LP64 that gained the greatest popularity on initial steps of development of 64-bit systems and now is used widely in Unix world. In this data model long type has size 8 bytes and it means that the code is absolutely correct. But in 64-bit Windows systems data model LLP64 is realized where long size remains 4 bytes and consequently this code will be incorrect. One should use, for example, LONG_PTR type or ptrdiff_t type in Windows.

Fortunately, the given code will be diagnosed as dangerous one by Microsoft Visual C++ 2005 compiler itself and by Viva64 analyzer as well. But you should always remember about such traps while using static analyzers.

We have an interesting situation. The question of porting programs on 64-bit systems was discussed thoroughly and different methods and rules of checking in static analyzers were carried out, and after that this theme became uninteresting. Many years have passed, many things have changed but the rules according to which the analysis is fulfilled remain unchanged and unmodified. It is difficult to explain what is the reason for this. May be, developers just do not notice changes thinking that the question of testing and checking of 64-bit applications has been solved long ago. I would like you not to be trapped in such a way. Be careful. That which has been actual 10 years ago may not be the same now, on the other hand a lot of new things have appeared. While using static analysis means make sure that they are combinable with the 64-bit data model you use. If the analyzer does not meets the necessary demands make some effort to find a different one and make up the gap by using specific Viva64 analyzer. The efforts spent on this will be compensated by the increase of safety of your program, the reduction of the terms of debugging and testing.

Now let's talk about unit tests. We will also face some unpleasant troubles concerning them on 64-bit systems. Trying to reduce the time of taking tests, developers usually tend to use a small size of calculations and the size of processed data while developing these tests. For example, developing a test with the function of searching an item in an array, it does not matter if it will process 100 items or 10.000.000. Hundred items will be enough but in comparison to the processing 10.000.000 items the speed of test passing can be significantly quicker. But if you would like to develop full tests to check this function of a 64-bit system you'll have to process more than 4 billions items! It seems to you that if the function works with 100 items it will work with billions? No. If you do not believe me try the following example on a 64-bit system to make sure.

```
bool FooFind(char *Array, char Value,
             size_t Size)
{
  for (unsigned i = 0; i != Size; ++i)
    if (i % 5 == 0 && Array[i] == Value)
      return true;
  return false;
}

#ifdef _WIN64
  const size_t BufSize = 5368709120ui64;
#else
  const size_t BufSize = 5242880;
#endif

int _tmain(int, _TCHAR *) {
  char *Array =
    (char *)calloc(BufSize, sizeof(char));
  if (Array == NULL)
    std::cout << "Error allocate memory" << std::endl;
  if (FooFind(Array, 33, BufSize))
    std::cout << "Find" << std::endl;
  free(Array);
}
```

As you may see from the example, if your program on a 64-bit system starts processing a large data size, you should not rely on old sets of unit tests. You should extend them surely taking into account processing of large data sizes.

But unfortunately, it is not enough to write new tests. Here we face the problem of the speed of fulfillment of the modified tests set, which covers the processing of large data sizes. The first consequence will be the impossibility to add such tests into the set of tests launched by a developer in the process of development. It may be problematic to add them into night tests too. The total time of passing of all tests may increase in an order or two, or even more. As a result the test may last even more than 24 hours. You should keep it in mind and approach the modification of tests for a 64-bit version of your program seriously.

The way out in this situation is to divide all the tests into several groups which are taken on several computers simultaneously. You can also use multiprocessor systems. Of course it will complicate the testing system in a way and demand more device resources but it will be the most right way and a simple step to solve the task of building unit testing system finally. Surely you'll have to use the system of automatic testing which will allow you to organize the launch of tests on several computers. The example is the system of testing Windows applications AutomatedQA TestComplete (http://www.automatedqa.com). With its help you can fulfill the distributed testing of applications on several workstations, carry out synchronization and results gathering [12].

At the end I would like to return to the question of white-box testing method which we considered to be inadmissible for large systems. We should also add that while debugging large arrays this method becomes more inadmissible. Debugging of such applications may take much more time and be difficult for using on the developer's computers. That's why one should think over the possibilities of using ordering systems for debugging applications or use other methods. For example, it can be remote debugging.

To sum it up, I would like to say that you should not rely only on some particular method. A quality application may be developed only with the use of several of the approaches to testing we've observed.

Summing up the problems of developing 64-bit systems, I would like to remind you the key moments:

1. Be ready for unexpected troubles while developing and testing 64-bit applications.
2. Be ready that the debugging of 64-bit applications with the help of white package may become impossible or very difficult if large data arrays are processed.
3. Study thoroughly the possibilities of your static analyzers. If it does not meet all the necessary demands make some effort to find a different one or use an additional static analyzer like Viva64.
4. You should not rely on old sets of unit tests. It is necessary to look them through and add some new tests which take into account the peculiarities of 64-bit systems.
5. Remember about significant reduce of unit tests speed and take care of providing new computers to launch them in time.
6. Use a system to automate testing which supports separate launch of applications like TestComplete system, and that will provide a quick check of applications.
7. The best result may be achieved only when you use a combination of several different methods.

The author hopes that this article will be useful in your work and wishes you successful release of your 64-bit projects. If you found a mistake in this article or would like to add something, the author will be glad to get your letters and opinions. I wish you successful work!

References

1. Wikipedia, "Code review", http://www.viva64.com/go.php?url=49
2. Steve McConnell, "Code Complete, 2nd Edition" Microsoft Press, Paperback, 2nd edition,

Published June 2004, 914 pages, ISBN: 0-7356-1967-0.
3. Wikipedia, "Static code analysis", http://www.viva64.com/go.php?url=31
4. Scott Meyers, Martin Klaus "A First Look at C++ Program Analyzers.", 1997, http://www.viva64.com/go.php?url=13
5. Walter W. Schilling, Jr. and Mansoor Alam. "Integrate Static Analysis Into a Software Development Process", 01, 2006, http://www.viva64.com/go.php?url=33
6. Wikipedia, "White box testing", http://www.viva64.com/go.php?url=47
7. Wikipedia, "Black box testing", http://www.viva64.com/go.php?url=48
8. Wikipedia, "Unit testing", http://www.viva64.com/go.php?url=44
9. Justin Gehtland, "10 Ways to Make Your Code More Testable", July, 2004, http://www.viva64.com/go.php?url=50
10. Paul Hamill, "Unit Test Frameworks", November 2004, 212 pages, ISBN 10: 0-596-00689-6
11. Andrew Josey, "Data Size Neutrality and 64-bit Support", http://www.viva64.com/go.php?url=51
12. AutomatedQA, "TestComplete - Distributed Testing Support", http://www.viva64.com/go.php?url=52
